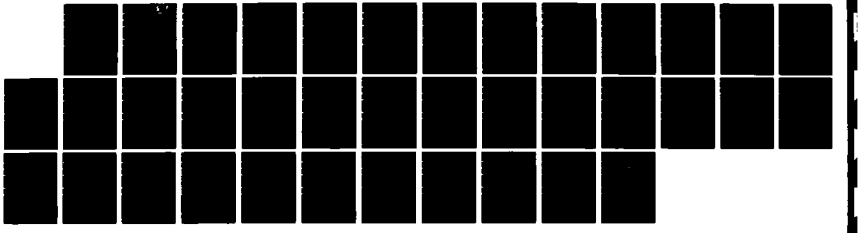
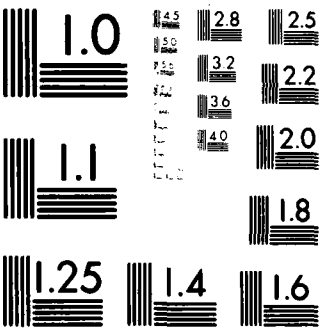
AD-A170 113 PORTABLE OPERATING SYSTEMS FOR NETWORK COMPUTERS: 1/1
DISTRIBUTED OPERATING S... (U) STATE UNIV OF NEW YORK AT
STONY BROOK DEPT OF COMPUTER SCIENC... L D WITTIE
UNCLASSIFIED 31 OCT 85 ARO-18864.12-EL DAAG29-82-K-0103 F/G 9/2 NL





SECURITY CLASSIFICATION OF THIS PAGE

ORT DOCUMENTATION PAGE

1a. AD-A170 113

1b. RESTRICTIVE MARKINGS

2a.

3. DISTRIBUTION/AVAILABILITY OF REPORT

DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE

4. PERFORMING ORGANIZATION REPORT NUMBER(S)

5. MONITORING ORGANIZATION REPORT NUMBER(S)

ARO 18864.12-EL

6a. NAME OF PERFORMING ORGANIZATION: SUNY-Stony Brook

6b. OFFICE SYMBOL (If applicable):

7a. NAME OF MONITORING ORGANIZATION: U.S. Army Research Office

6c. ADDRESS (City, State and ZIP Code):
Department of Computer Science
State University of New York
Stony Brook, New York 11794-4400

7b. ADDRESS (City, State and ZIP Code):
Research Triangle Park
North Carolina 27709-2211

8a. NAME OF FUNDING/SPONSORING ORGANIZATION: U.S. Army Research Office

8b. OFFICE SYMBOL (If applicable):

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAG-29-82-K0103 / P-18864-EL

8c. ADDRESS (City, State and ZIP Code):
Research Triangle Park
North Carolina 27709-2211

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

11. TITLE (Include Security Classification): Portable Operating Systems (unclassified)

12. PERSONAL AUTHOR(S): Prof. Larry D. Wittie

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM 4/12/82 TO 8/31/85	10/31/85	37

16. SUPPLEMENTARY NOTATION

Subtitle: Distributed Operating Systems Support for Group Communications

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Distributed Operating System, Hierarchy, Management Computer Network Communication

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The project "Portable Operating Systems for Network Computers" that ran from April 1982 until August 1985 saw the creation of an entire distributed operating system (MICROS/SAM2S) for a multiple ethernet system of DEC LSI-11 and Motorola 68000 systems. The modularity, hidden type managers, abstract manager hierarchies, message-oriented drivers, and layered tasks used in the design of the SAM2S operating system allowed the easy porting of the original LSI-11 based system to Motorola 68000 processors and its extension to a plexus of four interconnected thernets. The system was used to explore ways to support efficient communications within groups of processes scattered over many computers, especially in very large networks. This project has carefully delineated different classes of multicast communications within networks and has shown three different efficient ways to implement multicast: host, channel, and tree-based protocols. Only tree multicast is suitably efficient for networks of thousands of computers.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT

UNCLASSIFIED/UNLIMITED [X] SAME AS RPT. [ ] DTIC USERS [ ]

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Prof. Larry D. Wittie	(516) 246-8215	

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

Portable Operating Systems for Network Computers: Distributed Operating Systems Support for Group Communications

Prof. Larry D. Wittie
Computer Science Department
State University of New York
Stony Brook, New York, 11794-4400
(516)246-8215

The project "Portable Operating Systems for Network Computers" that ran from April 1982 until August 1985 saw the creation of an entire distributed operating system (MICROS/SAM2S) for a multiple ethernet system of DEC LSI-11 and Motorola 68000 systems. The modularity, hidden type managers, abstract manager hierarchies, messsage-oriented drivers, and layered tasks used in the design of the SAM2S operating system allowed the easy porting of the original LSI-11 based system to Motorola 68000 processors and its extension to a plexus of four interconnected ethernets. The system was used to explore ways to support efficient communications within groups of processes scattered over many computers, especially in very large networks. This project has carefully delineated different classes of multicast communications within networks and has shown three different efficient ways to implement multicast: host, channel, and tree-based protocols. Only tree multicast is suitably efficient for networks of thousands of computers.

This is the final report on the project "Portable Operating Systems for Network Computers" that ran from April 1982 until August 1985. With the support from the Army Research Office (DAAG-29-82-K0103), this project saw the creation of an entire distributed operating system (MICROS/SAM2S) for a multiple ethernet system of DEC LSI-11 and Motorola 68000 systems. This operating system was used to explore ways to support efficient communications within groups of processes scattered over many computers, especially in very large networks of computers. Besides the research and the development of the operating system, this work has seen the completion of two PhD dissertations and twelve Masters projects and the publication of nine papers[1-9] in the area of distributed systems. A list of the fourteen students with their dissertation and project titles is appended in the final acknowledgement section.

The next section of this report describes the design principles for SAM2 and the organization of two SAM2S implementations. The following two sections give the status of SAM2S and MICROS and conclusions reached in using Modula-2 to develop and port SAM2S. The last half of this paper expounds a theory of multicast communication facilities appropriate for large networks and tells the results of adding group communication primitives to MICROS/SAM2S.

1. Introduction to MICROS/SAM2S

The MICROS project is exploring ways to organize networks of thousands of computers (network computers) to solve large problems. Its main goals are to develop a portable distributed operating system (MICROS) that can efficiently control many different network computers and to produce cost-effective network computers that provide high-throughput for large classes of applications, that extend easily to form more powerful systems, and that are always available to users at acceptable processing rates even after component failures.

A network computer consists of many computer nodes, each with its own primary memory, physical clock, and attached peripherals. Nodes are embedded in a network of communication links over which messages are exchanged to share data from the separate memories. A global decentralized operating system, with some code resident in every node, unifies the nodes into a single computer system. The global operating system strives to provide network computer users with a powerful computing facility that can be accessed as a single virtual multiprocessor without regard to physical locations within the network.

Modula-2[10] is a high-level, general programming language that facilitates the building of simple and practical programming support systems. The Stand-Alone Modula-2 System (SAM2S) is a portable, highly-modular concurrent operating system. SAM2S was developed initially to assess Modula-2 as a language for writing large systems and to provide portable software for Modula-2 programming support workstations. SAM2S was first developed for DEC LSI-11 workstations and later ported to Heurikon/Motorola 68000 workstations. When replicated in every node of a network computer, SAM2S forms the locally resident portions of the MICROS distributed operating system.

2. Stand-Alone Modula-2 System (SAM2S)

The originally released Modula-2 system (M2RT11)[11] is a Modula-2 programming support environment targeted for DEC PDP-11 and LSI-11 systems and dependent on DEC's RT-11 operating system for services such as file access, editing, and I/O handling. The MICROS research group developed the first single-machine version of the Stand-Alone Modula-2 System (SAM2S) for the LSI-11 by writing standard Modula-2 library modules for all the RT-11 services used by M2RT11. SAM2S was first developed mainly to see if Modula-2 was adequate for producing entire operating systems for programming support workstations. It has proved more than adequate. The original version of SAM2S actually runs slightly faster than M2RT11, primarily because all service routines are kept resident by SAM2S and not paged from disk as for RT-11.

The small memory (60 KB) addressable by the LSI-11 limits the size of tasks run under the LSI-11 version of SAM2S to about 30 KB. In practice, this means that we could edit and compile simple modules under the LSI-11 version of SAM2S, but had to rely on M2RT11 to change large modules such as the passes of the compiler itself. Since the two systems run on the same processor with exactly the same file format, switching from one to the other required only a single 'boot' command. The lack of memory space on the LSI-11, especially as we began to write and test communication software, led us in 1983 to port SAM2S to workstations based on Motorola 68000 processors.

SAM2S has been designed to provide both a stand-alone programming support environment and a module library that can be the basis for the locally resident portions of the decentralized MICROS operating system. Each SAM2S instance on a single processor is a concurrent system, but not a distributed one. However, the SAM2S design emphasizes flexibility in communication whether on one machine or many and includes ethernet drivers, Xerox communication protocols[12] and DoD standard TCP/IP protocols.

2.1. SAM2S Design Principles

SAM2S is a highly portable, independent Modula-2 programming support environment based on a modularized kernel task running on a process-multiplexed microcomputer. The design for SAM2S uses many advanced features of Modula-2. SAM2S benefits heavily from high-level device drivers and from modularization facilities that allow definition of hidden and hierarchical type managers as well as layered tasks for both system and users.

2.1.1. Hidden Type Managers

The existence of a module facility does not automatically assure software modularity. Some programming standards are needed. For example, SAM2S code avoids both exported variables and nested modules. Module structuring in SAM2S is based on abstract data types, encapsulation concepts, and information hiding principles[13, 14]

A module should be designed to encapsulate one abstract data *type*, which imposes modular structure on data and characterizes all allowed operations and values. Each instance of a type is referred to as an *object*. The procedures in a module that define all operations on an object collectively form the *type manager*. Basic operations include creation, manipulation, and destruction of objects.

Hidden types in Modula-2 are declared only by name in the type definition module. The component substructure for the type is fully declared only in the implementation module. Hidden type objects are completely encapsulated. Only operations defined by their type manager can access or change them. Other modules do not know their structures and cannot directly manipulate their components. That hidden objects must contain all their own state information also allows their type manager to synchronize accesses efficiently. Process blocking is reduced by enforcing synchronization on individual shared objects only, rather than on the shared manager itself, as is done using monitors.

2.1.2. Hierarchical Type Managers

Two goals of type manager design are simplicity and generality. Simplicity demands a small module with a clean and readable structure. Generality means that each type manager should support an elaborate type with widely useful operations. These two goals usually are in conflict. Both goals can be achieved using policy/mechanism separation[15] and hierarchical type managers.

With policy/mechanism separation, lower levels of the system focus on providing general mechanisms that are as devoid as possible of embedded control decisions, so

higher levels have maximum flexibility in choosing policies. Type managers should be designed to adhere to the *type policy* determined by indicators within the object state. Their mechanisms must accomodate all allowed type policies.

With hierarchical type managers, a first-level manager handles the basic version of a general type. A second-level type manager uses the facilities of the first-level manager to offer more advanced operations and to support an extended type. Even higher-level managers may be defined. An example is a process type manager which provides basic operations like create, suspend, and resume. A more advanced manager uses additional information in each process object for synchronization.

2.1.3. High-Level Message-Oriented Device Drivers

Physical and logical devices can be regarded as hidden types requiring storage access, data transfer, and synchronization facilities. Physical device drivers manage the details for peripheral devices. Logical device modules support available I/O formats for character and block-oriented devices and interact with physical device drivers. Each device module is an active type manager since it contains one or more processes for device handling and user interactions. In SAM2S, the only processes that are genuinely concurrent are physical device processes that do real I/O by using the IOTRANSFER mechanism. All other processes are preemptively multiplexed by a time-slicing scheduler.

Device modules are written in high-level Modula-2 code, instead of assembly language, greatly easing system maintenance. Each device driver requires about 500 lines of Modula-2 code. Device drivers use low-level machine access facilities to manipulate device registers. Depending on the exact configuration of SAM2S, I/O service requests may be made directly through procedure calls, locally by interprocess messages using simple queue interfaces, or remotely through socket interfaces by messages from processes on other computers. Although the message interfaces for I/O are slower than direct entry procedures, they are extremely flexible and make it easy to reconfigure SAM2S for differing devices.

2.1.4. Layered Tasks

A *task*, or *concurrent program*, is a software structural unit built from one or more modules. Each task is a separate loading unit. Processes within a task are scheduling units that execute on a single host. Processes communicate and synchronize by passing messages and sharing objects. Linkers, editors, filers, and debuggers are common library tasks.

In Modula-2 systems, a task is specified by the hierarchy of module import dependencies that starts from the main module. The modules forming a task are linked together as an overlay onto a host. Normally, the operating system kernel forms the basis for all other task overlays. Other tasks are loaded in layers above it and access its modules by imported procedures. Where there is a system configuration choice of different implementation modules for the same type manager, one has to be specified. Linking the chosen modules automatically selects any library modules that they import. Modules that are needed by higher-level tasks, but have already been provided for

lower-level tasks are not linked again.

The main program module, base task, and selected module choices are presented to the SAM2S task linker to produce a relocatable load unit. The linker manages the module and task libraries, typechecks intermodule interfaces, and places the resulting load file in the task library. The file contains information for controlling task loading.

SAM2S supports the open system concept[16] which blurs distinctions between system and user tasks to enhance system flexibility.. The operating system is viewed as a collection of possible facilities that users can selectively include. Unneeded facilities cause no runtime overhead. All module interfaces are available to users. Hierarchical type managers allow users to select interfaces suitable for their application. Code for modules that are heavily shared among tasks is not repeated, reducing task sizes and increasing memory utilization.

2.2. SAM2S Organization

SAM2S runs on development systems based both on the DEC LSI-11/23 (SAM2S/LSI11) and the Heurikon HK-68K board version of the Motorola 68000 (SAM2S/68000). The LSI-11/23 workstations contain only 60 KB of memory, severely limiting task sizes for SAM2S/LSI11. However, there is no similar constraint on the Heurikon HK-68K workstations. Each currently has 256 KB to 768 KB of memory.

The overlay base for the SAM2S system is the highly modularized Kernel task. Most of its modules are hidden type managers. They are available to user tasks also. For SAM2S/LSI11, the Kernel task is generated by merging it with a language support subkernel of 1,000 assembly instructions that provide runtime trap handling and coroutine process primitives. For SAM2S/68000, the Kernel task contains the MC68000 module written mainly in Modula-2. This module provides runtime facilities similar to those of the LSI-11 assembly subkernel. The Modula-2 CODE procedure is used to generate about 800 lines of assembly code at specific points in the MC68000 module.

The following sections describe kernel modules in functional groups. Figure 1 shows the groups of modules in the SAM2S/68000 kernel task. Differences between the two SAM2S implementations occur only in the low-level kernel support modules and in the physical device drivers.

2.2.1. Kernel Support Modules

Low-level kernel modules are machine dependent. In SAM2S/LSI11, the LSI11 module encapsulates the architecture of the LSI-11/23 microprocessor. It defines machine specific trap and peripheral addresses that are also used by the assembly subkernel. The MC68000 module in SAM2S/68000 provides similar trap and peripheral access services. On each system, the Exceptions trap handling module is closely coupled to the basic trap facilities in the low-level machine module.

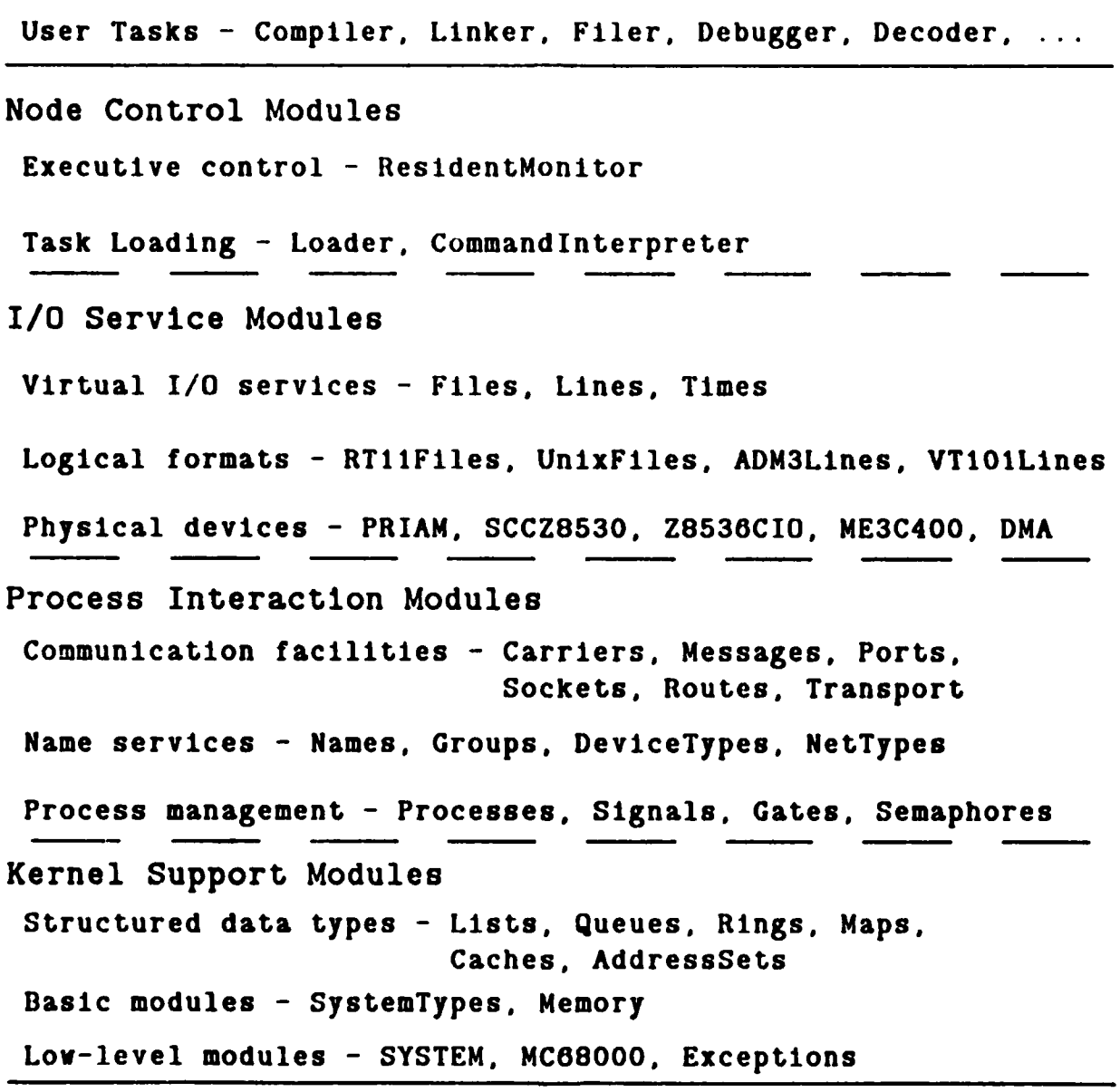


Figure 1: Structure of the SAM2S/68000 Kernel Task

The SystemTypes module exports basic constant and type declarations used throughout the system. Grouping common declarations into a single module lessens the number of interfaces that have to be imported by most modules. Memory management, including compaction, is provided in the kernel by the Memory module. Available memory is managed as a dynamic heap using a circular first-fit algorithm.

The structured data type modules are hidden type managers for abstract data structures needed by the kernel and by user tasks. For example, the Lists module can efficiently manage LIST objects created as a regular list, a descending or ascending priority list, a circular list, or a stack. The Maps module manages MAP objects, which are dynamically varying lists that associate an index for a hidden object with a unique identifier. Sets and caches of network communication addresses are maintained by the AddressSets and Caches modules. Other structured data types include queues and character buffer rings.

2.2.2. Process Interaction Modules

Process interaction facilities are provided by a hierarchy of type managers. The Processes module provides the basic PROCESS type and standard operations including process creation, blocking, resumption, suspension, and termination. Priority lists are used for process scheduling. Spawning of processes forms tree hierarchies used for process control and termination. Processes can be synchronized by use of the Signals, Gates, and Semaphores modules. Signals are events or conditions on which processes can wait and about which they can be notified. A SIGNAL object manages a list of processes queued on the associated event. A GATE object is used as a binary semaphore to support mutually exclusive access to shared objects or code sections. It can be used to implement monitors. More elaborate synchronization can be achieved with the general SEMAPHORE type that provides conditional blocking of processes. Other synchronization types include event counts and sequencers.

The Names and Groups managers provide services for registration and lookup of symbolic names. The NAME type associates the name string for an object with its attributes, access capabilities, and unique identifier. A capability contains addressing information and possibly object access rights. To provide for hierarchical name spaces, groups of names are managed in tree directories. The GROUP type supports none, one, or more associated NAMEs. Symbolic names can be searched for on the top level of any specified subtree or recursively throughout the subtree.

Communication facilities are provided by another hierarchy of managers. The Ports module uses Queues to support either First-In-First-Out (FIFO) or priority ports for sending and receiving local messages. It controls port access rights, message forwarding, and conditional passing of messages. For network communication, the NetTypes module declares common addresses and services. The Sockets type manager provides location-independent general message transfer services either locally, within the same host computer, or remotely, between processes on different hosts. A SOCKET is a bidirectional port used as an end-address for sending and receiving messages between processes. The Transport and Routes Modules provide for forwarding of packets over the communication subsystem.

To provide type uniformity for messages, Ports and Sockets directly manage carriers, which are standard headers for messages. Information in each carrier includes source and destination addresses, a unique message identifier, the message type, and a pointer to the message itself, if it exists. Empty carriers can be posted in ports for incoming messages. The Messages module provides packaging facilities for marshalling and unmarshalling data into and from packets used for remote procedure calls.

2.2.3. I/O Service Modules

I/O services are provided on three levels of abstraction: physical, logical, and virtual. Users interface at the virtual level for file and terminal services. The virtual level passes user requests as procedure calls or messages to the appropriate I/O format module on the logical level. The logical level interfaces with the appropriate physical I/O driver by messages using either communication or queue services.

The DeviceTypes module declares constants and types used by the physical and logical device modules. At initialization, device drivers configured in the kernel task register their existence with the name manager to give users dynamic access to I/O services. Device modules request I/O services and post results by using the IOREQUEST type as a standard message.

Examples of physical drivers for SAM2S/LSI11 QBUS-based devices include a DEC DLV11J serial driver, RX02 and RP02 disk controllers, and a QE3C400 ethernet controller. The DLV11J driver manages up to four serial lines and provides type-ahead terminal handling using a RING object for a character buffer. The RX02 floppy disk controller handles two diskette drives. The RP02 module handles eight logical partitions of a 169 MB Fujitsu Winchester disk. QE3C400 interfaces to one or two 3COM ethernet boards used for network computer communication.

Functionally similar physical drivers exist for MULTIBUS-based devices in SAM2S/68000. The SCCZ8530 module drives the Zilog serial communication chip on the Heurikon HK-68K board. It also handles up to four serial lines. SAM2S/68000 has a controller for a Priam 70 MB winchester disk, for several other disks (Vertex, Micropolis) and for the four direct memory access (DMA) ports on the Heurikon board. The DMA module supports efficient copying of blocks of data for both disk and ethernet facilities. ME3C400 provides a dual ethernet interface for SAM2S/68000.

Logical device modules include handlers for serial terminals and disk formats. The logical devices are independent of actual physical interfaces. The RT11Files module handles RT11 directory and file formats. A UnixFiles module could be substituted to handle Unix format files. ADM31Lines and VT101Lines control ADM31 and VT101 terminals.

The virtual level modules provide abstract services to their clients. The Times module provides time and timeout facilities using the KW11L and Z8536CIO physical clock drivers in SAM2S/LSI11 and SAM2S/68000, respectively. The Files and Lines modules provide an abstract file and serial line interface for users. These modules direct user requests to the proper logical device modules.

2.2.4. Node Control Modules

The ResidentMonitor executive module receives control after kernel initialization and monitors the execution of user tasks. It interacts with the command interpreter and kernel loader to load, execute, and terminate relocatable user tasks on SAM2S. At present, a single user code file at a time may be run. The file name serves as a load command.

2.2.5. SAM2S User–Level Tasks

Additional library modules are available for tasks run at the system user–level. Some allow changes to file names and options. File I/O can be abstracted into character I/O by using Streams. The Strings module supports standard operations, such as extract and concatenate, on strings represented as character arrays. InOut provides transparent access to characters on either files or terminals. SAM2S supports a five pass Modula–2 compiler, a task linker, link and load file decoders, a mini–core debugger, a static symbolic debugger, a filer, an import dependency charter, and other utility tasks.

3. SAM2S/MICROS Network Status

The Stony Brook network computer is based on Motorola's MC68000 and Digital Equipment Corporation's LSI–11/23 hosts interconnected by 10M–bps Ethernet channels (Figure 2). The nine existing hosts are used as programming support workstations controlling one or two terminals (designated *T* in the figure) each. Each MC68000 host has .25M or .75M bytes of memory, but each LSI–11 has only 64K bytes. There are four Ethernet channels in the configuration. Each host is connected to two channels at most.

Some hosts have Winchester disks (designated *W*) and some dual floppy disk drives (designated *F*), but two MC68000 hosts (8 and 9) have no attached disks. The flexibility of interfaces in Micros allows diskless MC68000 hosts to be booted remotely with files supplied from a disk on another MC68000. Individual application programs can be remotely loaded into any of the MC68000 hosts. One MC68000 (host 4) controls a color monitor (designated *M*) that shows Ethernet traffic among network computer hosts on its two directly–connected channels. Packet glyphs move nearly in real time, with just enough slowing for humans to see. The Vax–750/Unix system on channel 1 is used for cross development, object downloading, and remote file transfers through host 3.

SAM2S has successfully been ported from DEC LSI–11/23 computers to the Heurikon/Motorola 68000 single board systems within the Stony Brook network computer. On the ethernet links between network nodes, SAM2S uses flexible communication techniques including location–independent sockets, remote–procedure–call interfaces for file services, standard Xerox Network System (XNS) packet transport protocols[12] and most recently the DoD standard TCP/IP protocols for network communication.

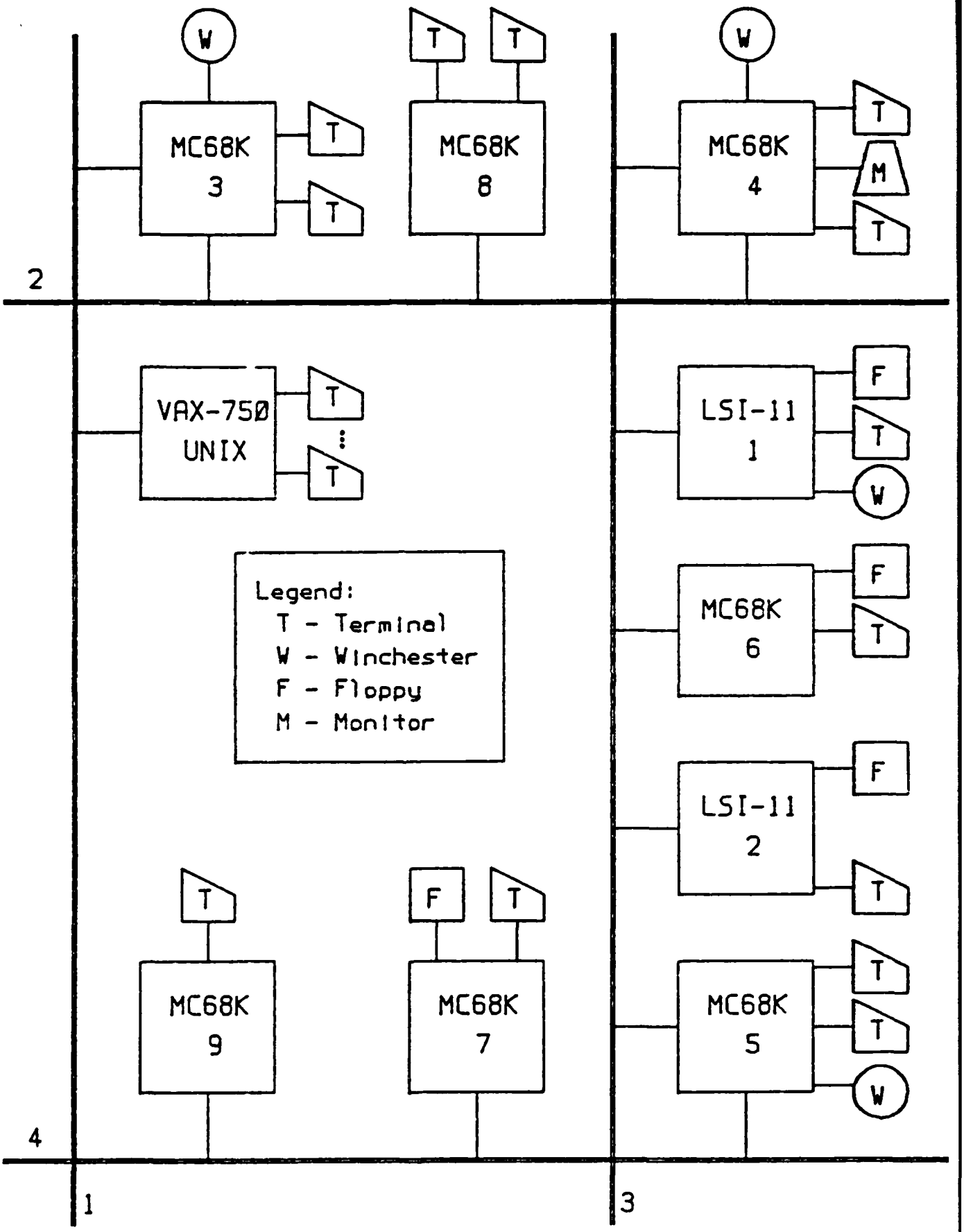


Figure 2. Stony Brook netcomputer of four Ethernets

Besides the original LSI-11 compiler from Wirth at ETH-Zurich and a VAX/VMS Modula-2 system from the University of Hamburg, there are several locally developed Modula-2 compilers that are being used to port SAM2S to other machines. The most heavily used is a VAX/UNIX cross-compiler that produces 68000 machine code. A translation of this compiler from Pascal into Modula-2 allows compilation directly on SAM2S/68000 systems. There is also a Modula-2 cross-compiler system running on VAX/UNIX systems and generating code for the Intel 8086 and 80186 processors.

The original version of MICROS[17, 18] was a modular, distributed operating system written in Concurrent Pascal[19] and assembly code. It ran on a network of DEC LSI-11 systems. With the addition of network communication modules and remote services between nodes, SAM2S has become the local operating system portion of MICROS. This new version of MICROS is written completely in Modula-2 except for a few hundred lines of assembly code. Its design emphasizes portable, transparent control structures. Control in MICROS is decentralized and distributed[20, 21] throughout the system as groups of cooperating tasks.

The new MICROS system contains more than 100,000 lines of local code. Except for the cross-compilers, almost all is written in Modula-2. The local operating system kernel, support and communication modules for SAM2S/LSI11 consist of 23,000 lines of code; similar modules for SAM2S/68000 take 27,000 lines. About 18,000 lines are identical in the two systems. The common but different 5,000 lines handle low-level system features and drivers for the almost disjoint sets of peripherals. The extra 4,000 lines in SAM2S/68000 are mainly a hardware-level debugging monitor for the 68000 processor and the more extensive network communication modules that the larger 68000 memory allows. The working cross-compiler for the 68000 and its translation into Modula-2 together take about 40,000 lines. The linker, loader, filer, editor, and other user-level system programs require about 9,000 lines. Each SAM2S system has about 7,000 lines of machine-dependent modules in its compiler, linker, and loader. There is another 20,000 lines in the code-generation passes of the compilers for the 80186, VAX-11, and 16000 processors. In addition, there is about 30,000 lines in LSI-11 compiler, linker/loader, and debugging utilities obtained from Wirth. More than 50,000 lines of high-level code were added to MICROS in 1984 alone.

4. Conclusions from Porting SAM2S

We have found Modula-2 much better for writing system code than the combination of Concurrent Pascal[19] and assembly code that we used for the first version of MICROS during 1978-81. The Modula-2 system, running on the same LSI-11 processor, is faster by a factor of 4 to 10 in several modalities. The 68000 version is even faster. Compiler and system code run faster since native machine code, not interpreted P-code, is produced. Flexible, selective synchronization operations defined by library modules allow faster execution of highly concurrent systems than do Concurrent Pascal monitors, which block processes too indescriminately. System errors can be located much faster using the post-mortem symbolic debugging system that is part of the Modula-2 task library. System corrections are faster because only a few modules, not the entire system, must be recompiled for each set of corrections, since there is type-checked, separate compilation of Modula-2 modules. System development by a

group is faster since only definition modules providing the interfaces between modules need to be approved before all programmers can start producing and compiling code.

The tiny runtime system, small compiler, and use of device interfaces written in high-level code all greatly simplify the porting of Modula-2 systems. We did not encounter major problems in porting SAM2S to 68000 systems. A few high-level modules have been changed slightly to make them truly machine-independent. Almost all the changes have involved the consistent use of long and short variants of integers and cardinals on the two systems. Communication between heterogeneous computers requires an external standard for the order of byte transmission. We have chosen the Xerox ethernet standard of high-byte-first order, as it also is for the 68000 microprocessor. Bytes are reversed in order as they enter or leave any of the LSI-11 systems. Porting SAM2S to a new computer requires rewriting of about 7,000 lines for code generation and loader modules, 1,000 lines for low-level kernel modules, and 1,000 to 4,000 lines for new device drivers.

Use of Modula-2 allowed us to port SAM2S from LSI-11 to 68000 systems in six months. It has allowed us to combine the efforts of dozens of student programmers into a working operating system. Since ADA is essentially a superset of Modula/*-2, the design features that allowed easy porting of SAM2S should work equally well for ADA systems.

5. Systems Research Using MICRO

A modular, integrated group communication subsystem has been implemented within MICROS to provide a basis for construction of a complete MICROS network computer system. The MICROS system must work well both on different types of computers and on networks that are connected in different ways - ways not known while the MICROS software is being written. Distributed control algorithms already designed for MICROS have included a Focus[22] initializer that transparently forms of a network-wide hierarchical control structure and a distributed Wave Scheduler[23, 24] that assigns idle nodes to task forces. The wave scheduling technique relies on a control hierarchy, includes mechanisms for avoiding static deadlocks, and can extend to any size network. We also have begun evaluation of other decentralized algorithms for management of globally shared system resources in large network computers with thousands of nodes.

Figure 3 shows one use of overlapping communication groups within a decentralized control hierarchy. Each triangular boundary encloses two groups. The working group consists of a number of sibling nodes plus their common parent. The recovery group adds the grandparent to the parent and siblings. The siblings execute user and management tasks as requested by the parent. To avoid overloading the parent during normal working conditions, the siblings pass to their parent only task results and summaries of management information about lower-level groups. If one of the nodes fails, the remaining members of the recovery group all communicate to redistribute the tasks of a failed sibling or to elect a replacement for a failed parent. Management information in a failed parent can be regenerated from the combined states of the siblings and grandparent. A failed grandparent is replaced as a parent by the next higher group in the hierarchy.

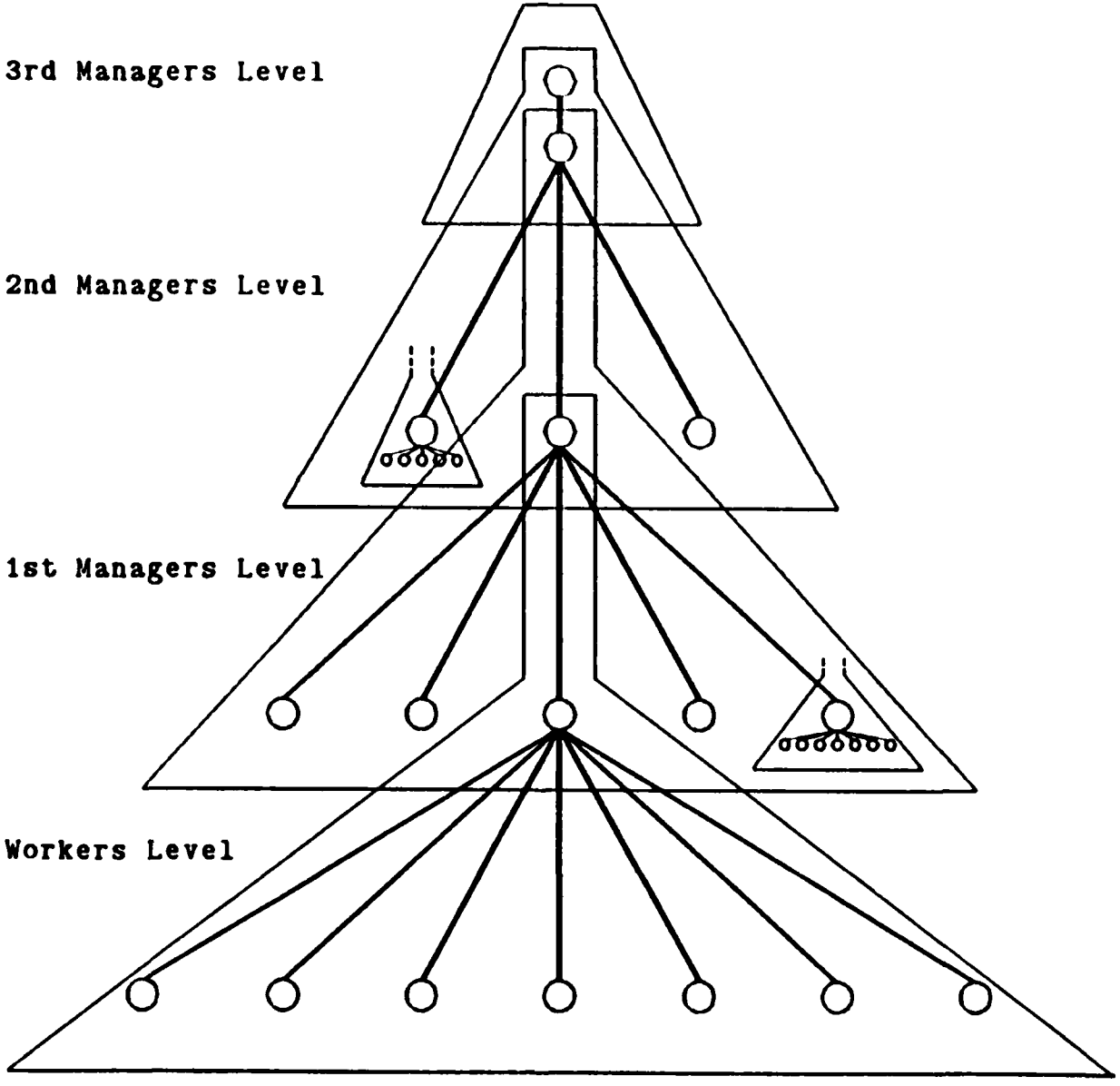


Figure 3: Hierarchical Tree Based on Dynamic Communication Groups

The next sections discuss efficient multicast mechanisms for sharing data among groups of processors or processes in large network computers. The major recent theoretical work[8] in the MICROS project has been in analyzing ways in which to implement and to use multicast communication within dynamic groups of computers in large networks, especially ones linked by grids of horizontal and vertical ethernets. Group communication techniques developed for ethernet systems should be applicable to many distributed system environments, even those using dedicated links. The research has included analysis of efficient network computer mechanisms to maintain distributed lists characterizing dynamically changing groups and to multicast packets within groups. Efficient communication in large groups can require spanning trees of multicast routing information. Single messages multicast to processes scattered over a network can follow tree branches and be copied at each fork.

6. Introduction to Multicast Communication

The concept of grouping processes to achieve goals is vital to distributed systems. System services for a group help to support communication and coordination among its members. Distributed groups are often organized to achieve parallel processing, increase data availability, reduce response time, share resources, or increase reliability.

Processes in a group often need to multicast the same message to all other group processes. Such messages include computation results, search bounds, state changes, votes, and updates to replicated data. Group members may need to multicast to the group several times during an extended interaction period. Such *group multicast* is more effective if some underlying multicast structure exists for the group. However, not all multicast techniques are effective group multicast techniques.

6.1. Packet–Switched Networks

The fundamental unit of information flow in a communication network is a packet. In general, packets are sent by *hosts*, or nodes, on communication *channels*, or links. A channel can be a point–to–point link or a multiaccess broadcast bus such as an Ethernet. Here, we consider the general model as a packet–switched network in which hosts can store packets and forward them on their connected channels. (In subsequent discussion of a specific network computer model, we refer to broadcast buses as channels.) Packets are transported across a network in datagram mode, that is, with a "best effort to deliver." Providing reliable transport of multicast packets is difficult because each packet must be acknowledged from a possibly unknown number of destinations. Both point–to–point and broadcast networks can be packet–switched.

A group of processes implicitly defines a host group consisting of all hosts on which the processes execute. A host group can be designated explicitly by a list of member addresses or implicitly by a logical group address. With lists, each group member must maintain a list of members so it can multicast to them. A membership list is dynamic or static depending on whether it can change. If a logical address is used, all group members must have network interfaces that will accept packets sent to the logical address. Each interface must recognize multiple logical addresses if its host is a member in several groups.

Table 1. Relative rating of criteria for single multicast

Multicast Technique	Multicast Criterion			
	Bandwidth	Delay	State	Computation
Flooding	gross	medium	nil	medium
Separate	high	high	high	low
Multidestination	medium	medium	high	high
Partite	medium	medium	medium	low
Single-Tree	low	medium	low	low
Multiple-Tree	low	low	gross	low

Table 2. Relative rating of criteria for group multicast

Multicast Technique	Multicast Criterion			
	Preparation	Maintenance	Failure	Scale
Flooding	nil	nil	nil	high
Separate	medium	medium	low	high
Multidestination	medium	medium	low	high
Partite	low	low	low	medium
Single-Tree	high	medium	medium	low
Multiple-Tree	gross	high	high	gross

6.2. Evaluating Multicast Techniques

Evaluation criteria for a single multicast include:

Bandwidth – The communication cost of the packet headers for a single multicast. It is the sum of the number of packets sent over all channels times the average size of their packet headers.

Delay – The time from the start of the multicast until the last packet copy is delivered. Packet delay per channel is assumed to be uniform. Because packet copies are sent in parallel, delay is a maximum, not a sum. Techniques that minimize delay tend to maximize bandwidth and vice versa.

State – The summed cost of storing the information that allows members to multicast to the group. It can include logical identifiers, lists of member addresses, or forwarding sublists forming previously built multicast structures. The state information should be bounded.

Computation – The processing cost for a single multicast. It includes calculation of intermediate destinations and update of multicast state information.

Evaluation criteria for group multicast include:

Preparation – The initial cost of distributing multicast information to all members. It may include building a structure to lower average cost per multicast.

Maintenance – The cost of adapting multicast information as members join and leave the group.

Failure – The cost of recovering from the failure of a network host or channel. Failures may require routing around failed components or repairing multicast structures.

Scale – The sensitivity of a multicast technique both to larger groups in a fixed–size network and to fixed–sized groups in a distributed system of increasing size. Scale should be at most proportional to the increase in size.

Tables 1 and 2 rate six types of multicast techniques against these criteria for single and group multicast, respectively. Most of these techniques, which include flooding, separate addressing, multidestination addressing, partite addressing, single–tree forwarding, and multiple–tree forwarding, are adaptations of their broadcast counterparts. All techniques are evaluated for a large multicast group.

The five *relative* values per criterion are nil, low, medium, high, and gross. Low, medium, and high ratings are always given to some technique. The nil and gross ratings are used only for exceptional cases. A "utopian" multicast technique would have all nil ratings.

6.2.1. Flooding

Flooding is a brute–force technique for packet multicast is to *broadcast* identical packet copies on all channels. Each receiving host forwards copies to all its other channels. Only group hosts keep a copy of the multicast packet.

This scheme is very simple. State, preparation, and maintenance costs are negligible and network failures have almost no impact. However, the very name of the technique reflects its major disadvantage: it floods the network with packets. The delay

rating is medium because heavy loads slow channel access. Bandwidth use is extremely high, since packets are duplicated on multiple channels. Bandwidth waste increases with network size, causing a high scale rating. Such gross bandwidth usage is not justifiable for multicast.

For flooding to be practical at all, packet lifetime must be limited to prevent endless duplication. Solutions include recording packet sequence numbers to prevent retransmission and discarding packets after a fixed number of relays. However, the bandwidth is always high.

6.2.2. Separate Addressing

Separate addressing is a obvious technique for multicast is to send a *separately addressed* packet to each destination. Each member maintains a copy of the entire group membership list, which is acquired during group preparation. A large group has a large membership list, so the state and scale ratings are high. However, network failure has little effect.

The major disadvantages of this technique are its high bandwidth and high delay. Several copies differing only in their destination addresses are often sent on the same channel. The multicasting host sends packet copies sequentially, so delay can be high. However, this technique is suitable for groups with few destination hosts.

6.2.3. Multidestination Addressing

This multicasting technique sends a few *multiply addressed packets*, for each multicast. Each packet header includes a subset of the destination addresses. When a packet arrives at any host, its destination addresses are apportioned among multiple copies. Destinations with the same route share the same copy. Packet copies are forwarded to all destinations.

Multidestination addressing is hard to support because of the need for variable sized packet headers. Because of address apportioning, this technique has a high computation rating and a medium delay rating. A minimal number of packets are sent, but bandwidth use is medium because of their large headers. The state, failure, and scale ratings equal those for separate addressing.

6.2.4. Partite Addressing

This method is a combination of the separate and multidestination addressing techniques. Host destinations are partitioned by some common addressing locality, say, subnets or channels. Separate packets are sent to each partition for final delivery to all local hosts. During group preparation, each member receives a copy of the partition list. This technique is especially useful for broadcast internets (multiple Ethernets), with hosts partitioned by their channels.

This technique is highly resilient against failures. It has a medium state rating, since all members list only the channel destinations. It uses medium bandwidth, since several copies to different channels may be sent on the same initial channels. The delay is only medium because the multicasting host builds and sends fewer packets than in separate addressing. Adding hosts to the group may not increase the number of channel

destinations, so the scale rating is medium. This technique is suitable for a group that resides on a small number of channels, even if there are many hosts.

6.2.5. Single– and Multiple–Tree Forwarding

This technique builds a spanning tree for the hypothetical graph of network hosts connected by channels and forwards packet copies along the branches. Each host member maintains and uses an image of only the local branches, making forwarding computations simple. Three main types of tree structures have been investigated: shortest path, minimum spanning, and centered.

A shortest path tree is one with the shortest possible path from the root host to any other tree host. Multiple shortest path trees[25] minimize both delay and bandwidth but have a gross state rating, since each member has a separate tree. The preparation rating is gross for multiple trees because they are difficult to build in a distributed manner. Maintenance can use the existing trees and is rated only high. Tree table space is proportional to the square of group size, so the scale rating is gross.

Another technique, called reversed–path forwarding,[25] simulates multiple trees without actually maintaining them by using routing tables and two additional lists.[26] However, this technique may not deliver some packets if routing tables change during forwarding.

A minimum spanning tree[25] has the least total branch cost of all spanning trees. A centered tree[26] is a shortest path tree rooted at a host "in the center" of the group. A single tree minimizes both bandwidth and state but has a medium delay rating because not all paths may be minimal. However, a centered tree has only marginally better ratings than a minimum spanning tree. The preparation in building a single tree is high, but maintenance is medium and scale is low. Although sensitive to failures, trees form an excellent structure for group multicast. In general, trees have the lowest bandwidth and delay ratings. Trees are particularly suitable for point–to–point networks.

The major problem with host tree forwarding on broadcast internets is that separate packet copies are sent to each host, not to each channel. In addition, a spanning tree specifies intermediate hosts between member hosts. Allowing the use of alternate paths between members would be less failure sensitive.

A good solution to both problems is to span a logical tree over channels and not over hosts. Packets can be forwarded to channels and then to hosts, as in partite addressing. The tree spans only member channels; physical paths between them are decided by runtime routing. (The use of channel–based trees is further described later.)

7. The Network Computer Framework

A network computer is physically distributed like a local broadcast internet but provides its users a single virtual computer like a multiprocessor. It has three types of communication resources: channels, hosts, and sockets (Figure 4). A multiaccess broadcast bus with short transmission delays is referred to as a *channel*; a *host* is a processing node; and a *socket* within a host is a process address. There are physical and logical addresses for each resource type. A physical identifier refers to a specific instance of a communication resource; a logical identifier refers to a group of them.

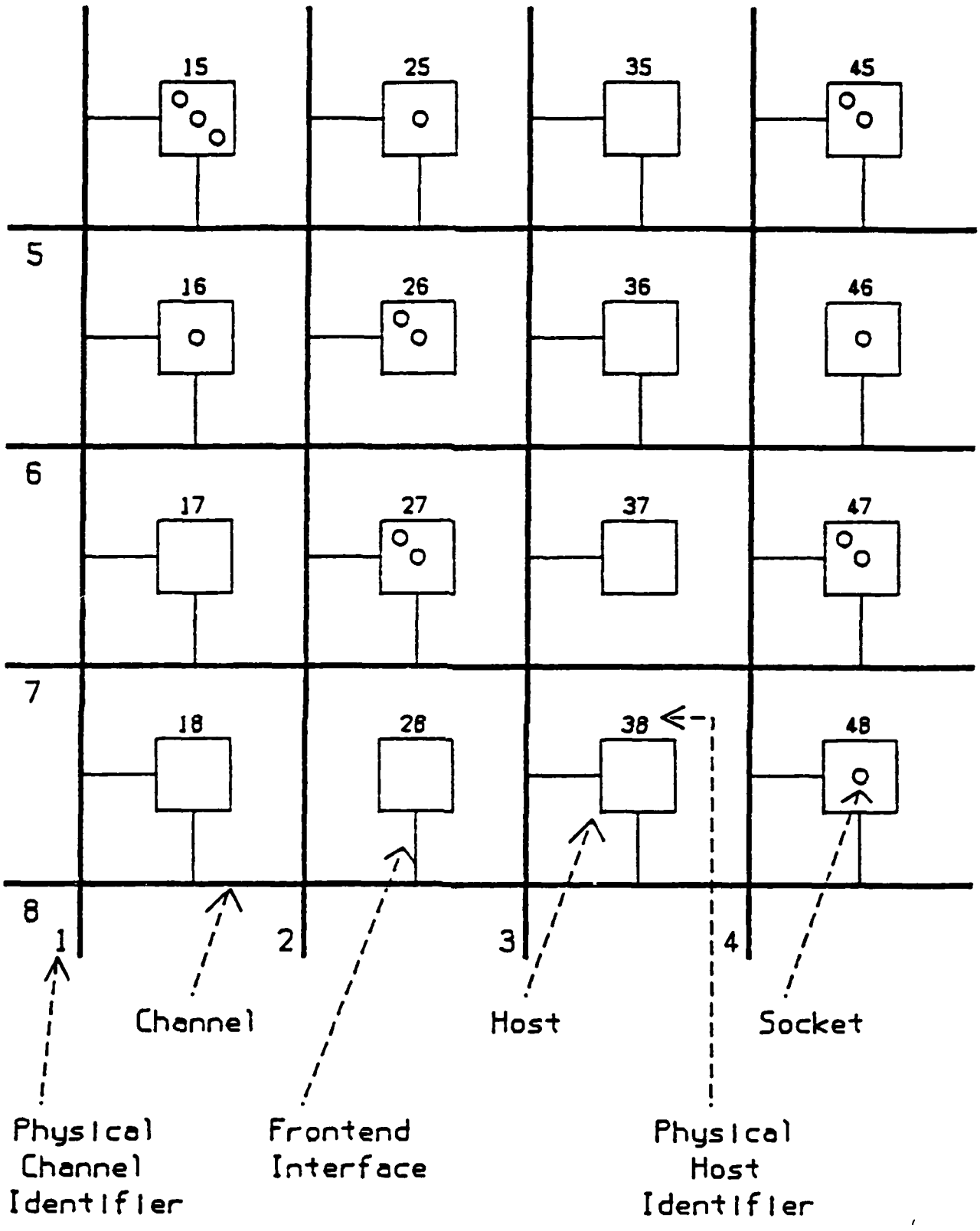


Figure 4. An example of a netcomputer

A broadcast bus supports logical addressing of multiple hosts. Network computer addressing conventions, based on those for internets in Xerox network systems,[27] extend logical host addressing over the entire network computer and provide for logical channel addressing. Each network computer address consists of channel, host, and socket identifiers. Since variable–sized packet headers can cause severe buffering problems, each packet header is assumed to have only one destination address.

7.1. Channels

A network computer can contain a large number of channels, such as Ethernets.[28] Each channel has a unique, 32-bit *physical channel identifier* in an unambiguous flat addressing scheme. Physical channel identifiers are used mainly as designators for network computer routing. The physical channel identifiers in Figure 4 are arbitrary.

A channel interfaced to a host via a transmission front–end is said to be *directly connected* to that host, otherwise it is *distant*. In Figure 4, channel 8 is directly connected to host 18, and channel 3 is distant. A channel can be directly connected to many hosts; a host can have several directly connected channels. Figure 4 depicts a grid network computer in which each host has at most two directly connected channels, nominally vertical and horizontal.

7.2. Hosts

A network computer can contain a large number of hosts. Each host has a unique, 48-bit *physical host identifier*. Network computer physical host identifiers are absolute and implement a flat addressing scheme that is independent of the channel addressing scheme and can be used in the generation of other unique identifiers. Each host gives its physical identifier to all its attached front–ends as its host address. In Figure 4, each physical host identifier is conveniently chosen as the concatenation of its two physical channel identifiers.

7.3. Sockets

A socket represents a bidirectional port, within a host that serves as a source and destination for packets. Packets can be both delivered to and transmitted from a socket. A host can support a large number of sockets. Each host can receive packets addressed to its sockets through all its directly connected channels.

Each socket has a 32-bit *physical socket identifier* that is globally unique, but ephemeral. It is generated uniquely by incorporating the unique part of a physical host identifier. For a 48–bit Ethernet address, this part is 20 bits long. A *logical socket identifier* designates a group of one or more sockets in one or more hosts. It is needed for all group members to receive a multicast packet on the same socket address.

8. Packet Cast

Since a network computer can be configured with an arbitrarily large number of channels and hosts, it can be quite large and the packet transport mechanisms quite complex. The transport of packets is done by a *network level* functionally comparable to the OSI network layer,[29] but without the strict interface boundaries. Packets are

transported across a network computer as datagrams.

Packet transport on a network computer is oriented mainly toward channels. Packets are transmitted on, switched among, and routed to channels. Routing to channels and not to hosts provides an order of magnitude reduction in the routing information required on each host.[27] Another order of magnitude reduction can be achieved by partitioning routing information among all hosts on each physical channel.

A single channel provides an excellent architecture for multidestination communication but has limited extensibility. Providing multicast communication on a network computer is harder, since packet casting on distant channels requires support. Requirements for packet casting on a network computer may be as simple as packet transmission to a single destination host on a directly connected channel or as difficult as transporting packet copies destined for a group of hosts on several distant channels. Packet casting is channel oriented, for efficiency. The type of packet casting is based on the location of the destination channel relative to the sender: physical, directed, or logical.

8.1. Physical Cast

Advanced transmission media such as Ethernet channels[28] directly support broadcast and multicast transmission. *Physical cast* is the transmission of a packet by a host on a directly connected channel and is triggered when the physical channel identifier of a directly connected channel is addressed.

8.2. Directed Cast

Physical cast provides for packet delivery only on a directly connected channel. There is also a need to direct a packet to a distant channel, which can be accomplished if the network level provides packet routing services. Each host can transport packets to a distant channel by physically unicasting them to an intermediate host for relay.

In *directed cast*, a packet is cast to one distant channel.[30] This type of packet casting is used for sending a packet whose destination address contains a physical channel identifier of a distant channel. Directed cast is a generalization of physical cast. It is done by a series of packet unicasts through intermediate hosts toward its final destination channel. Any host receiving the packet on the destination channel uses a final physical cast to deliver it to the destination hosts on that channel.

8.3. Logical Cast

Directed cast provides for packet delivery on only one channel. Multidestination communication sometimes requires casting packet copies on a number of physical channels. Here, the network level provides packet propagation, or *forwarding*, services. Forwarding services enable each host to propagate packet copies simultaneously to a set of physical channels.

Logical cast is a packet cast directed toward a local set of physical channels associated with a logical channel identifier given as a destination address. It results in a series of directed and physical casts of packet copies. A host performing a logical cast sends packet copies to the physical channels in its set of forwarding destinations. If a destination channel is distant, its packet is encapsulated for relay through intermediate hosts.

When a packet copy arrives at a host that is directly connected to a destination channel, a physical cast is done.

Figure 5 shows a logical multicast from host 17 that involves a directed multicast to channel 2 via relay 27 and several physical casts. A socket marks each group host: 15, 17, 18 on channel 1; 26, 28 on 2; and 36, 46 on 6. The directed multicast from 17 to channel 2 starts with a physical unicast to relay 27, followed by a physical multicast on channel 2 to members 26 and 28. A physical multicast from 17 to channel 1 reaches all three members there. Host 16 also accepts the packet to forward it on channel 6 by a physical multicast that reaches 36 and 46.

9. Group Multicast Techniques on Network Computers

Since a network computer is also a packet-switched network, three group multicast techniques (separate addressing, partite addressing, and single tree forwarding) are also suitable. Flooding, multidestination addressing, and multiple tree forwarding are not acceptable because flooding has a gross bandwidth rating, multidestination addressing requires variable-sized headers, and multiple trees are too expensive to build and store.

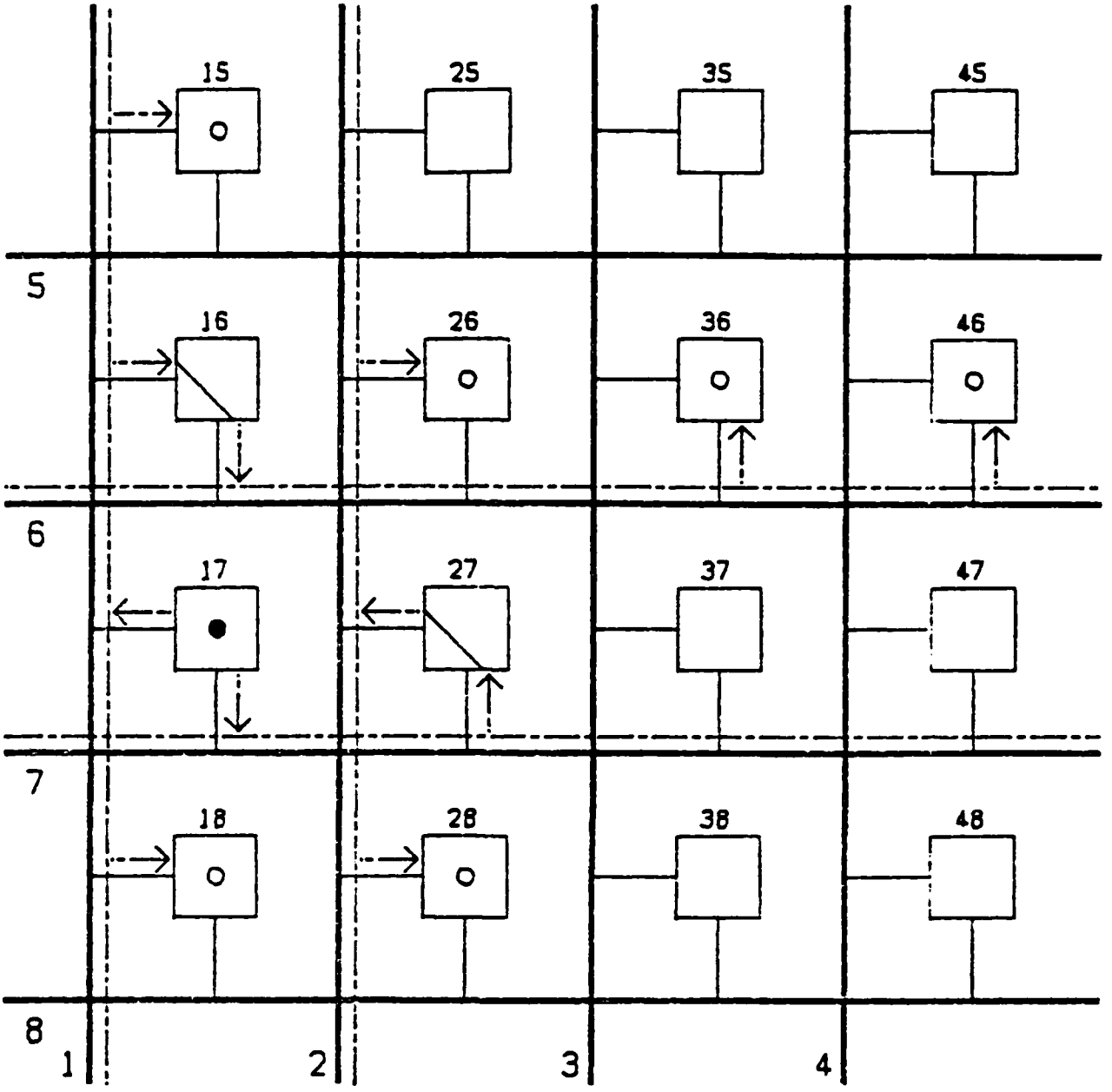
The *distance* from a source host to a destination host is the minimum number of directly connected channels forming a path between them. The distance between two hosts on the same channel is one; between a host and itself, it is zero. For example, in Figure 5, the distance between hosts 18 and 36 is two. The distance between a host and a destination channel is one more than the distance between the source host and the closest host on the destination channel.

The distance function is useful mainly for computing the bandwidth of a multicast. For packets with unit size headers, bandwidth is the sum of all channels traversed as part of a packet cast. The bandwidth of physical cast is one, since only one channel is involved. The bandwidth of a directed cast is the distance between the source and destination. The total bandwidth of a series of directed casts is the sum of the individual bandwidths.

For delay computation, each packet sent or received is assumed to cause a delay of one time unit. Each host that has two directly connected channels can send and receive two unrelated packets in one time unit. However, receiving and then sending a packet copy results in a delay of two units. Multicast delay computations assume optimal packet casting order. Packets are sent to destinations in decreasing order of distance.

9.1. Host Multicast

The separate addressing technique is used for host multicast. Processes communicate by multicasting to their group logical socket, which identifies the list of physical channel-host destinations. (In Figures 6, 7, and 8, the list is (4,46), (4,47), (4,48), (5,15), (5,25), (5,35), (7,27), and (7,37)). To multicast a packet, a separate packet copy is sent by directed unicast to each physical host in the list. Each destination address is composed of physical channel and host identifiers and the logical socket identifier. Each host in the group receives the packet destined to its physical host identifier and delivers it to the logical socket of the process group.



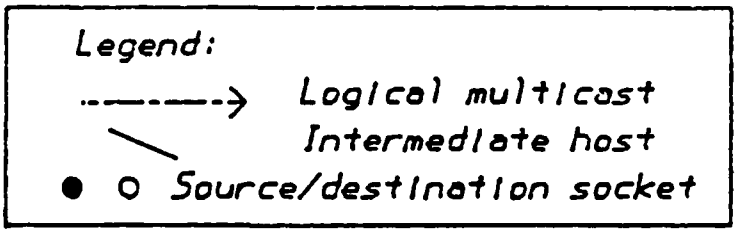
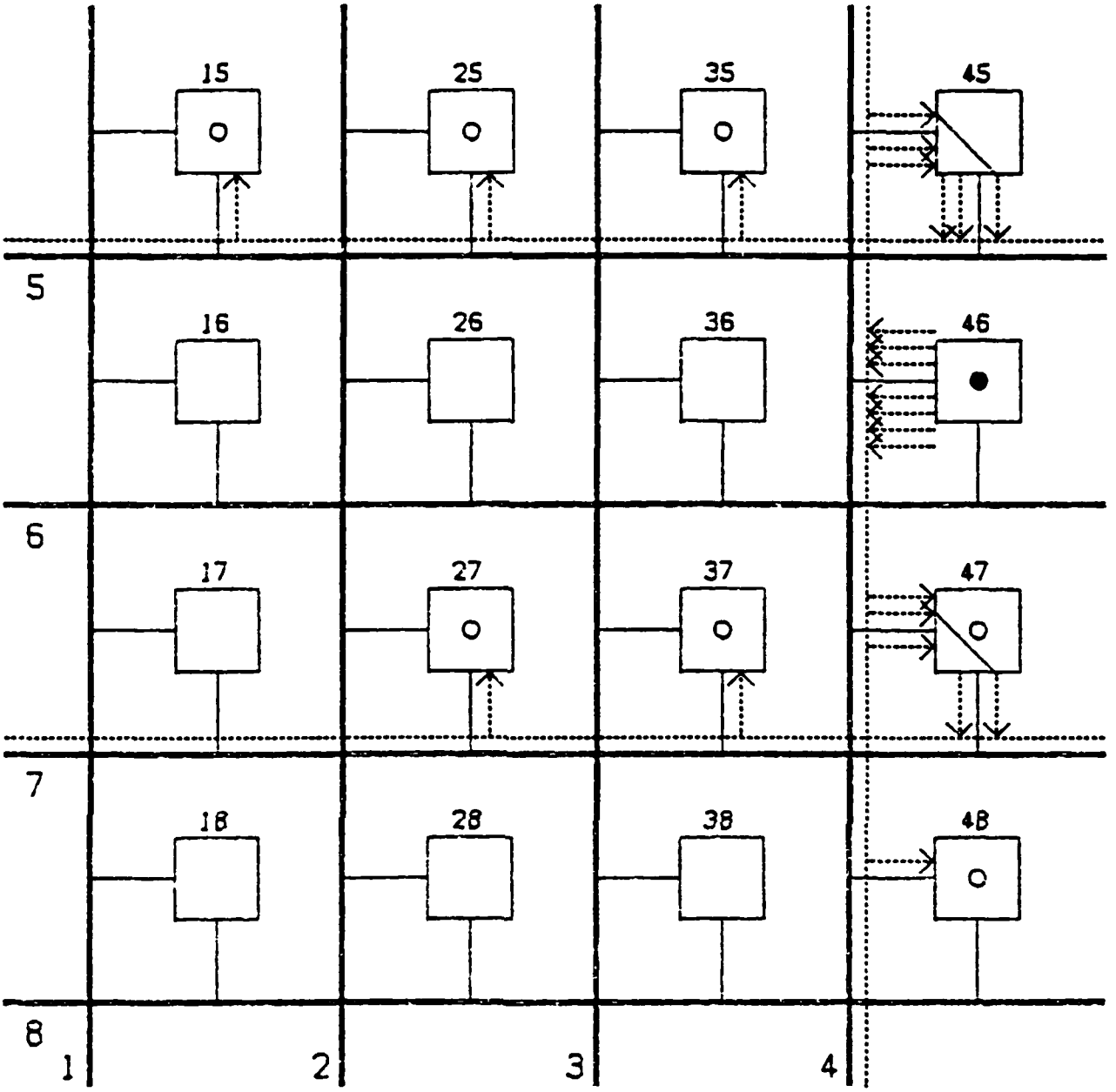


Figure 5. Illustration of logical multicast from host 17 to group hosts on channels 1,2,6



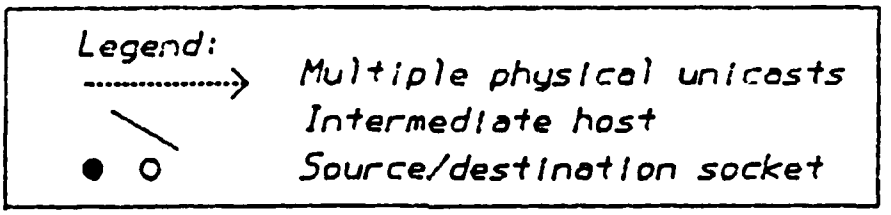


Figure 6. Example of host multicast from host 46 to group hosts on channels 4,5 and 7

In Figure 6, a multicast from host 46 to the group requires 12 packets. There are two physical unicast packets to hosts 47 and 48 on channel 4. Of five directed unicasts, two are to hosts 27 and 37 through intermediate host 47 and result in four packets. The other three directed unicasts to hosts 15, 25, and 35 through intermediate host 45 result in six packets. A maximum of 12 packets is needed for multicast from any host in the group for this example. Ten packets from host 47 are the minimum. In the example, the multicast delay is at best eight, since host 46 sends seven packets and the last packet must be received. The bandwidth and delay costs for host multicast in this example are thus 12 and 8.

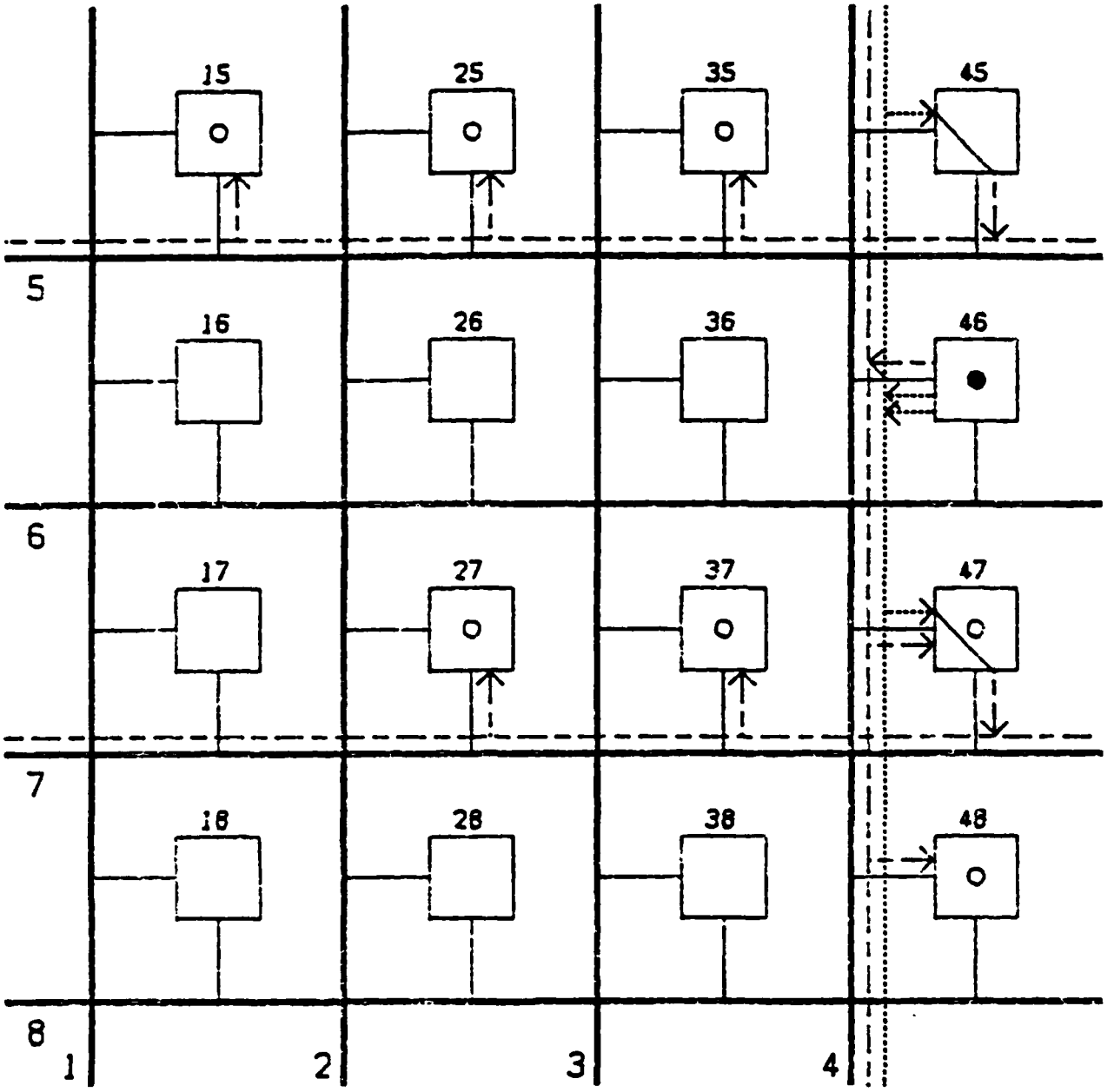
Host multicast is simple to maintain but expensive to use. Although it uses only directed and physical unicast, it has high state and bandwidth ratings. Host multicast is useful when each group member needs an explicit list of all members for other purposes, when the host group is small, or when the group exists for too short a time to justify building a more complex multicast structure.

9.2. Channel Multicast

The partite addressing technique is used for channel multicast. Each group host maintains only the list of physical channel identifiers on which all group hosts reside. (In Figure 7, the channel list is 4, 5, and 7.) Each host member must recognize packets addressed to the group's logical host identifier. To multicast a packet, a separate copy is sent to each physical channel in the list. Each destination address consists of the same logical host and socket identifiers but contains a different physical channel identifier. Physical multicast is finally used to deliver the packet to all group hosts on each group channel. Each host accepts the multicast packet and delivers it to the group logical socket.

In Figure 7, a multicast from host 46 to the group requires five packets, including one physical multicast on channel 4 and two directed multicasts. One directed multicast to channel 7 results in a physical unicast to host 47 and then a physical multicast on channel 7; the other multicast to channel 5 similarly uses two packets. Host 47 receives two packets: one for itself from the physical multicast on channel 4, and one for physical multicast on channel 7. At most five packets are needed and a minimum of four from host 47. From host 46 there is a delay of at least five to receive the last packet of the physical multicasts on channels 5 and 7. Both bandwidth and delay costs for this channel multicast are 5.

Channel multicast is simple and efficient if all group hosts are connected to only a few physical channels. In particular, it is best for a single channel since only one physical multicast is required. Channel multicast provides a more complex but efficient mechanism than host multicast. It is more complex because it uses directed and physical multicast. It is more efficient because only a list of all channels, not of all hosts, has to be maintained and used by each host in the group.



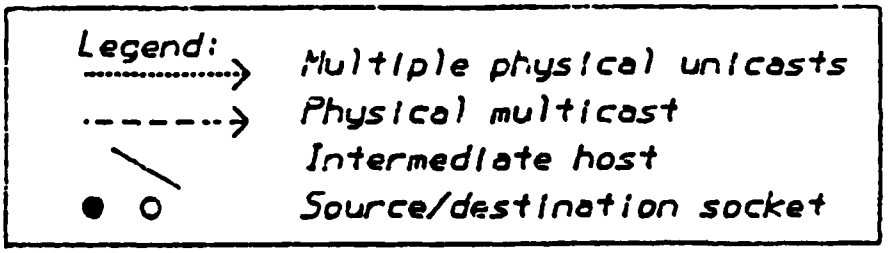


Figure 7. Example of channel multicast from host 46 to group hosts on channels 4,5 and 7

9.3. Tree Multicast

Logical multicast can be used to cast a packet on a previously built multicast structure, for example, a single shortest path tree. Logical trees are spanned over channels, not hosts. A logical tree is built using current route information about group physical channels. With tree multicast, each packet header contains one logical channel identifier, designating one or more physical channels.

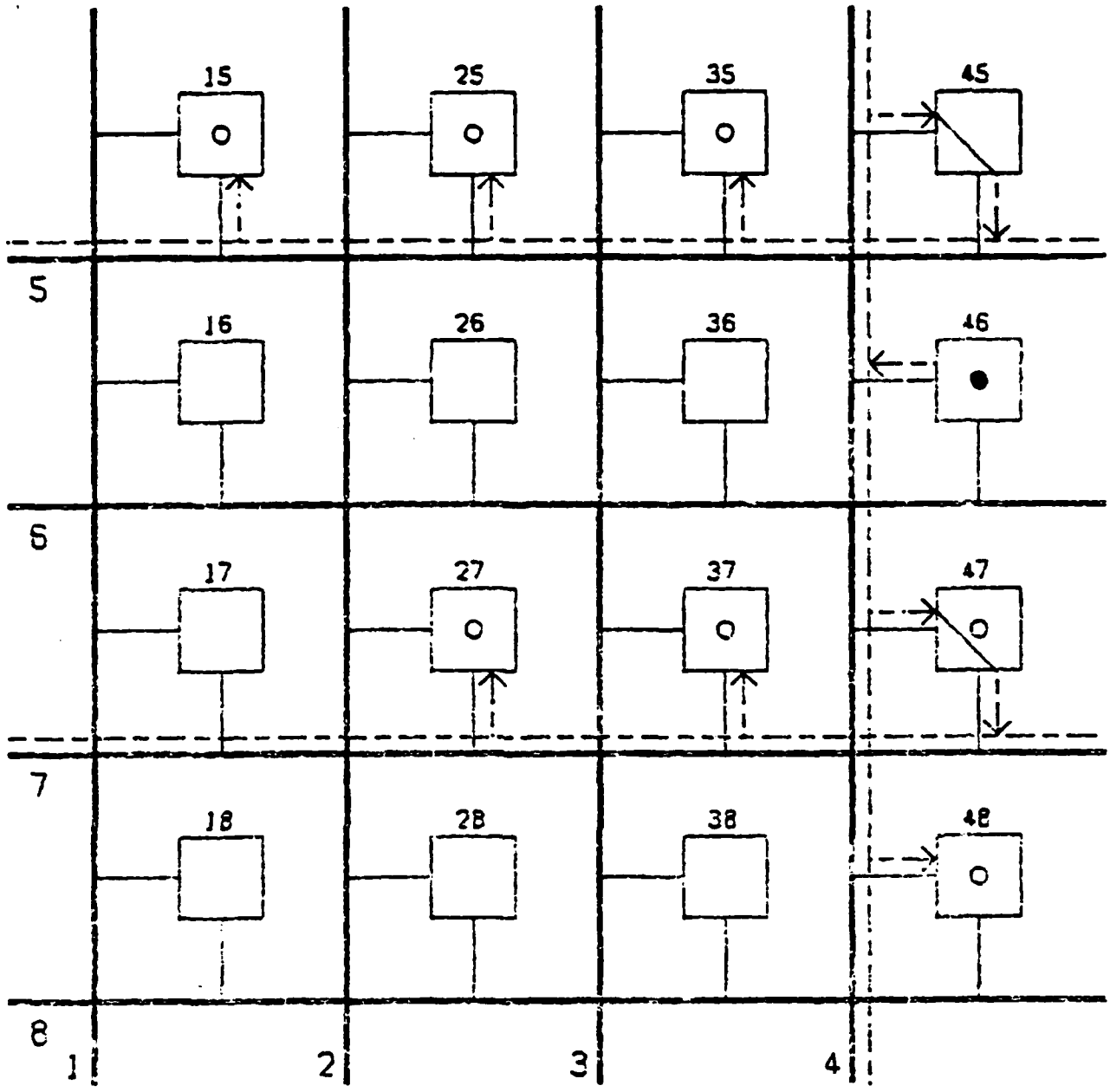
The principle of multidestination addressing can be used, for example, by a preparatory procedure to build a shortest path tree spanning the channel group. Using routing information, the initiating host creates several packet copies with the destination channel addresses partitioned among them as packet data. Addresses with the same intermediate route are placed in the same copy. Each outgoing packet is sent to the nearest channel of its subset.

To form the image of the local tree, forwarding services of hosts on each channel of the tree keep a list of all immediate destinations to which packets are forwarded. When each packet arrives at any host on its destination channel, multiple copies are again created to partition its destination addresses. Packet copies are forwarded until received by hosts on all original destination channels. Hosts left with no more destinations are on channels that form the leaves of the spanning tree. Hence, the tree building process ends.

Forwarding services on each host maintain the list of immediate parent and children channels that are the local image of the spanning tree. Tree multicast is accomplished by a series of logical multicasts of packet copies along the tree branches. Each packet destination address is composed of logical channel, host, and socket identifiers. Whenever a host on a destination physical channel receives a packet copy on one branch of its local subtree, it sends copies to the other branches of the subtree. When directed unicast is needed, the packet copy is encapsulated with a header for the physical channel destination and forwarded to an intermediate host. Eventually, the packet is delivered on all physical channels in the group.

In Figure 8, channel 4 is the tree root and channels 5 and 7 are its children. Hosts on channel 4 have the list 5 and 7. Hosts on channels 5 and 7 have the list 4. A multicast from host 46 to the group requires only three packets. There are three physical multicasts: one on channel 4 and two on channels 5 and 7, done by hosts 45 and 47, respectively. Three packets are the minimum and maximum needed. The multicast delay from host 46 is four since each packet copy to channels 5 or 7 needs to be sent, received by an intermediate host, relayed, and received on each final destination. The costs of tree multicast are thus a bandwidth of 3 and a delay of 4.

Tree multicast is useful for long-lasting groups with members scattered over many physical channels. For an extended communication period, time is well spent in efficiently connecting group members. This type of multicast is more complex but more efficient than channel multicast. The network level must build and maintain a tree, but a full channel list need not be maintained on all group hosts. Only local forwarding information is needed. Tree multicast is also more efficient because only a minimal number of packet copies are sent for each multicast. Since the tree is spanned over channels and known to all hosts on these channels, it is failure resilient.



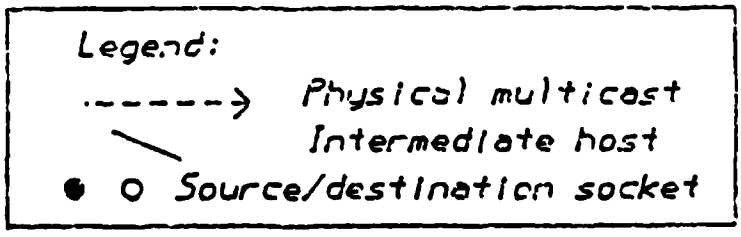


Figure 8. Example of tree multicast from host 46 to group hosts on channels 4, 5 and 7

10. Group Communication within MICROS

An experimental group communication subsystem has been integrated into the MICROS operating system[3, 7] on the Stony Brook network computer. The underlying objective of the MICROS project is to explore control and communication techniques for viable network computers with thousands of hosts. The MICROS research testbed is mature enough to provide a practical environment for studying distributed algorithms, languages, and applications.

10.1. Communication Subsystem

The communication subsystem consists of 10 modules that provide packet cast services and support Xerox-like packet transport protocols. Another 5 modules for DoD TCP, IP, and related auxiliary protocols have also been added for compatibility with Berkeley Unix 4.2+ communications. The ports module uses queues to support either first-in, first-out or priority ports for sending and receiving local messages. It controls port access rights, message forwarding, and conditional passing of messages. The sockets module provides location-independent message transfer services either locally within the same host computer or remotely between processes on different hosts.

To provide type uniformity for messages, ports and sockets directly manage carriers, which are standard headers for messages. Information in each carrier includes source and destination addresses, a unique message identifier, the message type, and a pointer to the message itself, if it exists. The messages module provides packaging facilities for marshalling and unmarshalling data into and from packets.

For network communication, the network types module declares common addresses and services. The routes module on each host maintains a cache of local routing information similar to that of the routing information protocol of the Xerox network systems. The forks module on each host maintains a cache of forwarding information for the logical channels recognized by the host on each of its directly connected channels. The transport module receives all the packets passed by its host or received by its front-ends. It uses the routes and forks services to do the packet cast appropriate for the packet destination address.

10.2. Group Subsystem

We assume that a group membership list exists during the group multicast period. Maintaining membership lists for dynamically changing groups is not an easy task. As part of the MICROS project, research was done on organizing dynamic groups.[5] A dynamic group is a decentralized group that is coordinated mainly using asynchronous messages. All members of such a group maintain a dynamically varying list of group members that can be used to support group multicast and distributed dictionaries of replicated data. Precise algorithms for maintaining dynamic groups have been developed, proved effective, and tested[31].

The group subsystem provides services for organizing dynamic groups of hosts and multicasting within them. Each group is associated with a logical address to enable multicast to its members. The communities module provides for the organization of dynamic groups and supports the three types of group multicast (host, channel, tree) by

using communication subsystem services. It uses the views module to maintain membership lists for all groups in which the host is a member. Members exchange views of the group to bring themselves up-to-date.

An interactive grouper program has been developed to test and demonstrate the maintenance of dynamic groups on the Stony Brook network computer. It allows the user to create and terminate groups. It permits queries about the status of groups maintained locally. For each group, the user can recruit and dismiss members, send membership list messages to other members, and multicast user messages to group members. Multicast messages received by the local host can be displayed on a terminal.

10.3. Performance Results

The major design goals for the SAM2S/MICROS kernel and communication subsystems were interface flexibility and ease of experimentation; performance was only incidental. However, several timing experiments have been done to compare the costs of the various interfaces. For reference, average process switching time on a MC68000 host is 2.5 ms.

When single-character serial I/O calls were timed on an LSI-11, the serving of local I/O requests through socket interfaces took about two to three times as long as through queue interfaces. As a compromise between speed and flexibility, sockets are standardly used for logical level I/O interfaces and faster queues are used for the physical level interfaces. A need for remote socket calls to low-level physical I/O drivers has not yet arisen.

Timing experiments were also done on communication services. Measurements for each operation were obtained by timing as the operation executed a 1000 times. For example, using the time of 56.6 seconds for 1000 messages, we can assume that sending a single message of 512 bytes through a socket would take 56.6 ms on average. The message and its carrier are copied into a packet by the socket manager, the route is determined by the transport manager, and the packet is delivered to the Ethernet driver. The packet is physically transmitted on the directly connected channel of the intermediate routing host. A self-addressed packet is sent last and retrieved to determine when all 1000 duplicated packets have been transmitted.

Unicasting an already prepared, 512-byte message directly through the Ethernet driver would take about 42.8 ms. The saving of an asynchronous interface to the transport manager to determine the route and update the carrier accounts for the 13.8 ms difference in the 56.6 and 42.8 ms times.

Similarly, the observed socket interface time to send a packet of 256 bytes was 42.5 ms; for 128 bytes, 36.0 ms; and for 64 bytes, 31.8 ms. The corresponding times to send a packet directly to the Ethernet driver were 28.2 ms for 256 bytes, 21.2 ms for 128 bytes, and 17.7 ms for 64 bytes. Calls to both interfaces caused each packet to be copied twice. The observed times to process a packet are about 14 ms each in the socket interface and in the Ethernet interface. Hence, about 0.028 ms is necessary to copy each byte of data.

Several experiments centered on the group subsystem and its use of communication services. Two groups of hosts were used (see Figure 2): 3, 4, 8 and 3, 4, 5, 8. Hosts 3, 4,

8 reside on channel 2 and 3, 4, 5, 8 on channels 2 and 3. Creating and sending a 168 byte membership list for the first group from host 8 to host 3 on channel 2 took about 55 ms. Since it takes 38 ms to send a 168-byte unicast packet, we can deduce that membership message preparation takes about 17 ms.

Host multicast and channel multicast have also been compared. A host multicast of two 64-byte packets from host 8 to hosts 3 and 4, all on channel 2, took about 68 ms. A channel multicast of one packet to both hosts 3 and 4 took only 42 ms. Because the 1000 multicast packets in each burst sometimes come too rapidly for the double buffers in each 3COM Ethernet interface, only about 65 percent of these burst multicast messages are actually received.

If host 3 is the multicast source to hosts 4 and 8, the corresponding times are 82 ms for host multicast and 50 ms for channel multicast. The times for host 3 to multicast to the group are higher than those for host 8 because host 3 has one more Ethernet interface than host 8. Host 3 multicasting to hosts 4, 5, and 8 in the second group took 115 ms and 85 ms. The times for the larger group are each about 35 ms longer because one additional packet is needed in each case to reach host 5 on the second channel.

These experiments were run with a small number of hosts and channels. The disparity between host and channel casting times would be even greater in a system with more hosts on each channel.

11. Dynamic Group Algorithms

One of the main uses of multicast communication is to spread important data among all members of a group of processes or processors that are cooperating on some common task. The task may be part of a user application part of or a system of related high-level managers for a distributed operating system. In a large distributed system, there needs to be some reliable way to organize groups of many processes, where the membership of the group changes over time. This is a dynamic group.

In work[5, 31] during the last part of this supported effort, we have been able to show how to organize dynamic groups so that they can agree upon a common membership list and on replicated user data. Even though each member may independent alter its local copy of the common list or data, the dynamic group algorithm is provably[31] guaranteed to maintain data consistency among members without synchronization delays and whether or not member processors fail. The lack of synchronization delays greatly speeds processing in very large networks.

Ensuring serializability and mutual consistency of replicated data is considerably more difficult in an asynchronous communication network where all local changes cannot be broadcast reliably to all members than in systems that take the time to get global approval for all pending changes. Distributed dictionaries are a useful database system for management of replicated data that need to be only weakly consistent. Weakly consistent means as consistent as the pattern of messages between distributed members allows. It means that whenever a message is sent between two members to check consistency, the local data copies held by each will be identical.

Dynamic groups can be organized by replicated dictionaries that are correctly maintained even over an unreliable network. The dynamic group algorithm continuously

keeps the group intact and all membership dictionary copies weakly consistent even though group information and time estimates are discarded soon after members are deleted from the group. Most dictionary messages can be asynchronous. Reliable handshakes are needed just for initially inserting a member and for finally discarding its group information. Dynamic groups can efficiently support the maintenance of replicated dictionaries for distributed user applications.

12. Summary

Support for this project has enabled the creation and porting of the SAM2S operating system for LSI-11s to a mixed network of LSI-11 and M68000 computers. It has provided a foundation on which to experiment with communication techniques suitable for very large networks of computers. Efficient ways to implement multicast communication and replicated data in large distributed systems have been devised. The next line of research within the MICROS project will be the exploration of global control techniques for networks of thousands of processors and simulating them by analysis and by testing them realistically on smaller physical networks. Experiments in distributed task force scheduling[24], in management hierarchy recovery[6], and searching via distributed name caches are planned. The results obtained from the MICROS research should be applicable to many similar distributed environments.

13. Acknowledgements

Special thanks go to the ARO Electronics Division administrators who have patiently helped further this work: Michael Andrews, Jim Gault, Ron Green, and the ever-present Jimmie Suttle.

This research with MICROS began at SUNY/Buffalo, building on an earlier hardware and software network computer built there. The dozens of students in Buffalo who helped include: R. Spanbauer, P. Bechtel, D. Benua, P. Henderson, J. Kolkovich, H. Styliades, and S. Wilder, for engineering, and A. van Tilborg, R. Curtis, A. Frank, W. Holmes, P. Lee, M. Palumbo, K. Tso, K. Wong, T. Bartkowski, A. Chin, G. Davidian, J. Day, W. Earl, S. Isaac, G. Masterson, D. Reif, P. van Verth, and R. Wahl, for software. Five of these people moved to SUNY/Stony Brook in 1982 to continue the work.

Many members of the MICROS research group helped to develop the SAM2S version of MICROS at Stony Brook. Especially important contributions have been made by Ariel Frank, Ron Curtis, Shridhar Acharya, Divyakant Agrawal, Bob Barkan, Benoy DeSouza, Miguel Garcia, Arun Garg, Stuart Jones, FanYuan Ma, Mike Palumbo, Yanick Pouffary, Shraga Schnitzer, Vipin Samar, Soumitra Sengupta, Aditya Singh, Rick Spanbauer, Shidan Tavana, Kok Sun Wong, and WeiMin Zheng.

The PhD and Masters degrees conferred from this work include:

(1) Andre M. van Tilborg - "Network Computer Operating Systems and Task Force Scheduling", Aug. 1982, PhD.

(2) Ariel Frank - "Distributed Dynamic Groups on Network Computers", Aug. 1985, PhD.

(3) Paul Lee, Aug. 1982, "Line and Screen Control for Modula-2 System"

(4) Kam-Sing Tso, Sept. 1982, "Memory Management System for Modula-2 MICROS/MICRONET"

(5) William S. Holmes, May 1983, "Version and Source Code Support Environment"

(6) Michael J. Palumbo, May 1983, "Stand Alone Modula-2 System Enhancement"

(7) Kok-Sun Wong, Aug. 1983, "Stand Alone Modula-2 System Relocatable Linker and Dynamic Loader"

(8) Yanick P. Pouffary, Aug. 1983, "The Network Layer Implementation of the MICROS Communication Subsystem"

(9) Miguel Garcia, Dec. 1983, "A Modula-2 Compiler for 68000 Systems"

(10) Arun Garg, Dec. 1983, "Transport Layer Implementation of the MICROS Communication Subsystem"

(11) Alice Tong, Dec. 1983, "Modula-2 Cross Compiler for the Intel 80x86"

(12) Patrick Lou, May 1984, "Comparison Class Programs in Pascal or Modula-2"

(13) Shraga Schnitzer, Dec. 1985, "The M2Debu Postmortem Symbolic Debugger"

(14) Benoy DeSouza, May 1986, "Implementation of IP on SAM2S for a Network of 68000s"

14. References

1. A. J. Frank, L. D. Wittie and A. J. Bernstein, "Multicast Communication on Network Computers", *IEEE Software*, **2**, 3 (May 1985), 49-61.

2. A. van Tilborg and L. D. Wittie, "Wave Scheduling – Decentralized Scheduling of Task Forces in Multicomputers", *IEEE Transactions on Computers*, **C-33**, 9 (September 1984), 835-844.

3. L. D. Wittie and A. J. Frank, "A Portable Modula–2 Operating System: SAM2S", *Proceedings AFIPS National Computer Conference*, **53**, (July 1984), 283-292.

4. R. S. Curtis and L. D. Wittie, "Global Naming in Distributed Systems", *IEEE Software*, **1**, 3 (1984), 76-80.

5. A. Frank, L. Wittie and A. Bernstein, "Maintaining Weakly-Consistent Replicated Data on Dynamic Groups of Computers", *1985 Proc. Int. Conf. on Parallel Processing (ACM)*, St. Charles, IL, August 1985, 155-162.

6. C. K. Mohan and L. Wittie, "Local Reconfiguration of Management Trees in Large Networks", *Proc. 5th Int. Conf. Dist. Comp. Sys.*, Denver, CO, May 1985, 386-393.

7. A. van Tilborg and L. D. Wittie, "Operating Systems for the Micronet Network Computer", *IEEE Micro*, **3**, 2 (April 1983), 38-47.

8. A. J. Frank, L. D. Wittie and A. J. Bernstein, "Group Communication on Netcomputers", *Proceedings 4th Intl. Conference on Distributed Computing Systems*, San Francisco, CA, May 1984, 326-335.

9. L. D. Wittie and A. van Tilborg, "An Introduction to Network Computers", *Proceedings ACM82 Conference*, Dallas, Texas, October 1982, 199-206.

Modula-2, Springer-Verlag, New York, NY, 2nd edition

chnical Report 36, Institut fur Informatik, ETH, Zurich,

rt Protocols", XSIS 028112, Xerox System Integration December 1981.

ria to be Used in Decomposing Systems into Modules", *CM*, **15**, 12 (December 1972), 1053-1058.

ftware for Ease of Extension and Contraction", *IEEE Engineering*, **SE-5**, 2 (March 1979), 128-137.

and S. P. Harbison, *HYDRA/C.mmp: An Experimental* w-Hill, New York, 1981.

. Sproull, "An Open Operating System for a Single-User *dings 7th Symposium on Operating System Principles*,

Tilborg, "MICROS, A Distributed Operating System for figurable Network Computer", *IEEE Transactions on* ecember 1980), 1133-1144.

Computer Operating Systems and Task Force Scheduling, t of Computer Science, SUNY at Buffalo, NY, September

Architecture of Concurrent Programs, Prentice-Hall, 977.

K. Dalal, "Techniques for Decentralized Management of *Proceedings IEEE COMPCON Spring 80*, February 1980,

alized Executive Control of Computers", *Proceedings 3rd tributed Computing Systems*, October 1982, 31-35.

D. Wittie, "High-Level Operating System Formation in *IEEE 1980 Intl. Conference on Parallel Processing*, August

D. Wittie, "Wave Scheduling: Distributed Allocation of rk Computers", *IEEE Proceedings 2nd Intl. Conference on Syst.*, April 1981, 337-347.

L. D. Wittie, "Distributed Task Force Scheduling in Networks", *Proceedings AFIPS National Computer* 1981), 283-289.

M. Metcalfe, "Reverse Path Forwarding of Broadcast *tions of the ACM*, **21**, 12 (December 1978), 1040-1048.

nisms for Broadcast and Selective Broadcast, PhD Thesis, boratory, Stanford University, June 1980.

rox Network System", *IEEE*

C. Crane, "Evolution of the ", **15**, 8 (August 1982), 10-27.

, Englewood Cliffs, NJ, 1981.

Stanford University, January lo Alto research center, Ca,

ork Computers, PhD Thesis, ook, NY, August 1985.

END

DTIC

8—86